Chapitre # (ANN) 4 Bonnes pratiques en Python

Résumé & Plan

Ce chapitre est transverse à tous les autres, et est d'une importance capitale pour la rédaction de vos projets. Le jury est attentif à tous ses éléments de présentation du code, ne le négligez donc pas. Il faut garder la chose suivante en tête : les examinateurs ont énormément de projets à lire pour l'oral, plus le vôtre sera clair et compréhensible, plus vous aurez de chance d'être compris. Pour ce faire, il y a un certain nombre de conseils simples à prendre en compte.

Code is read much more often than it is written.

— Guido van Rossum

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

— Martin Fowler

Comme vous l'avez constaté dans tous les chapitres précédents, la syntaxe de Python est très permissive. Afin d'uniformiser l'écriture de code en Python, la communauté des développeurs Python recommande un certain nombre de règles afin qu'un code soit lisible. Lisible par quelqu'un d'autre, mais également, et surtout, par soi-même. Essayez de relire un code que vous avez écrit « rapidement » il y a un 1 mois, 6 mois ou un an. Si le code ne fait que quelques lignes, il se peut que vous vous y retrouviez, mais s'il fait plusieurs dizaines voire centaines de lignes, vous serez perdus. Avec l'expérience, vous vous rendrez compte que cela est parfaitement vrai. Alors plus de temps à perdre, voyons en quoi consistent ces bonnes pratiques.

*Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex. Complex is better than complicated. Flat is better than nested. Sparse is better than dense. Readability counts. Special cases aren't special enough to break the rules. Although practicality beats purity. Errors should never pass silently. Unless explicitly silenced. In the face of ambiguity, refuse the temptation to guess. There should be one– and preferably only one –obvious way to do it. Although that way may not be obvious at first unless you're Dutch. Now is better than never. Although never is often better than *right* now. If the implementation is hard to explain, it's a bad idea. If the implementation is easy to explain, it may be a good idea. Namespaces are one honking great idea – let's do more of those!*

Un condensé de toutes les règles ci-dessous peut être consulté ici :
`https://larlet.fr/david/biologeek/archives/20080511-bonnes-pratiques-et-astuces-python/`

1. LA PEP8

Plusieurs choses sont nécessaires pour écrire un code lisible : la syntaxe, l'organisation du code, le découpage en fonctions, mais souvent, aussi, le bon sens. Pour cela, les PEP (comme « Python Enhancement Proposal »), qui sont des agrégats de propositions dont la plus connue est la PEP8. Voici une synthèse des principales préconisations.

1.1. Indentation

Déjà constaté en première année : l'indentation des programmes est un incontournable des scripts en Python. Cela vient d'un constat simple, l'indentation améliore la lisibilité d'un code. Dans la PEP8, la recommandation pour la syntaxe de chaque niveau d'indentation est très simple : **4 espaces**. N'utilisez pas autre chose, c'est le meilleur compromis.

Attention

Cela ne correspond pas nécessairement à la touche de tabulation de votre clavier, mais c'est le cas dans Pyzo.

1.2. Importations de modules

- Le chargement d'un module se fait avec l'instruction `import module` ou éventuellement avec un préfixe `import module as blabla` plutôt qu'avec `from module import *`.
- L'importation doit également se faire par ordre alphabétique de nom.

Si on souhaite ensuite utiliser une fonction d'un module, la première syntaxe conduit à `module.fonction()` ce qui rend explicite la provenance de la fonction. Avec la seconde syntaxe, il faudrait écrire `fonction()` ce qui peut causer divers problèmes (conflits notamment).

1.3. Règles de nommage

- Les `noms_de_variable` doivent être écrites en minuscules, avec éventuellement le symbole underscore comme séparateur de blocs de noms, avec un nombre de caractère petit (pour les lire facilement).
- Les `CONSTANTES` globales sont écrites en majuscules, avec éventuellement le symbole underscore comme séparateur de blocs de noms, avec un nombre de caractère raisonnable (plus facile à retenir).
- Nommage des variables globales : utiliser des noms explicites dans votre code plutôt que le codage utilisé.

Par exemple :

```
ma_variable
fonction_test_27()
mon_module
VITESSE_LUMIERE = ...
```

On essaiera toujours de donner des noms de variable explicites : il est complètement inutile de perdre de l'énergie à répondre en permanence à des questions du type « que représente telle ou telle variable »? Ainsi : dans le cas d'un plateau de jeu recouvert de pions blancs (codés par des 0) et noirs (codés par des 1), utiliser en préambule

```
NOIR = 1
BLANC = 0
```

et utiliser `NOIR, BLANC` plutôt que `1,0` dans tout le projet.

1.4. Espacement

- La PEP 8 recommande d'entourer les opérateurs (`+`, `-`, `/`, `*`, `==`, `!=`, `>=`, **`not`**, **`in`**, **`and`**, **`or`**...) d'un espace avant et d'un espace après.

Par exemple :

```
# code recommandé :
ma_variable = 3+7
mon_texte = "souris"
mon_texte == ma_variable
# code non recommandé :
ma_variable = 3+7
mon_texte = "souris"
mon_texte == ma_variable
```

Mais inutile d'aligner pour « faire joli ».

```
# code recommandé :
x1 = 1
x2 = 3
x_old = 5
# code NON recommandé :
x1    = 1
x2    = 3
x_old = 5
```

Et c'est tout. On n'espace pas entre un nom de fonction et son argument par exemple, dans l'indexage des éléments d'une liste, *etc.*.

- On met un espace après les caractères « : » et « , » (mais pas avant).

1.5. Aération & Retours à la ligne

AÉRATION. Dans un script, les lignes vides sont utiles pour séparer visuellement les différentes parties du code.

Il est recommandé de laisser deux lignes vides avant la définition d'une fonction ou d'une classe et de laisser une seule ligne vide avant la définition d'une méthode (dans une classe).

On peut aussi laisser une ligne vide dans le corps d'une fonction pour séparer les sections logiques de la fonction, mais cela est à utiliser avec parcimonie.

RETOUR À LA LIGNE. Pyzo matérialise une ligne sur la droite, qu'il convient de ne pas dépasser pour des questions de lisibilité. Bien entendu, comme sur l'image ci-dessous, des très légers débordements à droite peuvent être tolérés.

```
def Kmeans(cellule_1,cellule_2,distance):
    """
    Cette fonction sépare la liste de cellules en deux groupes à partir
    d'une cellule de type 2 et une cellule de type 4
    """
    Barycentre_1_i = cellule_1
    Barycentre_2_i = cellule_2

    Barycentre_1_j = calcul_barycentre(cellule_1,cellule_2,0,distance)
    Barycentre_2_j = calcul_barycentre(cellule_1,cellule_2,1,distance)

    while distance(Barycentre_1_j,Barycentre_1_i) > 1 and \
        distance(Barycentre_2_j,Barycentre_2_i) > 1 :

        Barycentre_1_i = Barycentre_1_j
        Barycentre_2_i = Barycentre_2_j

        Barycentre_1_i = calcul_barycentre(Barycentre_1_i,Barycentre_2_i,0,distance)
```

RÉGLAGE DE LONGUEUR DE LIGNES SOUR PYZO

- Tout code qui dépasse cette ligne doit être coupé, pour des questions de lisibilité évidente. [a]

a. Votre examinateur n'aura aucune envie de *« scroller »* de gauche à droite (en plus de haut en bas)

1.6. Commentaires

Les commentaires débutent toujours par le symbole # **suivi d'un espace**. Ils donnent des explications claires sur l'utilité du code et doivent être synchronisés avec le code, c'est-à-dire que si le code est modifié, les commentaires doivent l'être aussi (le cas échéant).

- Les commentaires sont sur le même niveau d'indentation que le code qu'ils commentent. Les commentaires sont constitués de phrases complètes, avec une majuscule au début (sauf si le premier mot est une variable qui s'écrit sans majuscule) et un point à la fin.
- Les commentaires qui suivent le code sur la même ligne sont à éviter le plus possible et doivent être séparés du code par au moins deux espaces.

Remarque 1 (Risques d'obscurcir le programme)

- En ajoutant du verbiage (paraphrase),
- en dispensant le programmeur d'écrire proprement,
- en induisant le lecteur en erreur.

Par exemple, est-ce pertinent d'écrire cela ?

```
x += 1 # on augmente x, COMMENTAIRE TOTALEMENT INUTILE
```

- Un commentaire n'est pas là pour expliquer une notion de Python, mais pour faire gagner du temps au lecteur de votre code. Il n'a de l'intérêt que s'il est nécessaire de devoir expliquer quelque chose à l'endroit donné.

Un commentaire peut être également utilisé pour expliquer, par exemple, les structures de données utilisées dans votre projet. Par exemple en début de projet :

```
# Les coordonnées sur l'échiquier seront représentées par
# des couples (i, j) (numéro de ligne, numéro de colonne), # la \
↪ numérotation commençant à zéro.
# On représente la solution en cours de construction par un # \
↪ tableau t de longueur au plus N donnant les coordonnées
# de chaque dame déjà placée.
```

2. AUTRES RECOMMANDATIONS

2.1. À propos des conditionnelles

SIMPLICITÉ DES `return`. La gestion des booléens doit être optimale. En particulier :

- Les booléens sont des objets comme les autres.
- `return x > 2` doit être aussi naturel et habituel que `return x + 2`.
- Ainsi, ce type de chose est à éviter :

```
if x > 0:
    return True
else:
    return False
```

Utiliser plutôt : `return x > 0`.

2.1.1. Imbrication

- Éviter d'imbriquer des conditionnelles. En particulier un `if` juste après un `if`, `elif` ou `else` doit être changé.

Par exemple :

```
if i==1:
if j==1: action(1) elif j==17: action(2) else: action(3)
elif i==17:
if j==1: action(4) elif j==17: action(5) else: action(6)
else:
if j==1: action(7) elif j==17: action(8) else: action(9)
```

doit **être changé en**

```
if (i, j) == ( 1, 1):
    action(1)
elif (i, j) == ( 1, 17):
    action(2)
elif i == 1 :
    action(3)
elif (i, j) == (17, 1):
    action(4)
elif (i, j) == (17, 17):
    action(5)
else:
    action(6)
```

3. LES DOCSTRINGS ET LA PEP 257

De manière générale, écrivez des docstrings pour les modules, les fonctions, les classes et les méthodes. Lorsque l'explication est courte et compacte comme dans certaines fonctions ou méthodes simples, utilisez des docstrings d'une ligne. Les éléments que vous allez renseigner dans la docstring sont d'importance capitale pour la personne qui va exécuter votre programme (le jury de projets pour vous en l'occurence). Elles comprennent notamment :

1. ce que fait la fonction ou la méthode,
2. ce qu'elle prend en argument,
3. ce qu'elle renvoie.

Il existe certaines règles pour la rédaction de ces aides, mais nous n'irons pas jusque là. Voici un titre informatif un exemple de « bon formatage » de docstring pour une fonction :

Version longue

```
def exemple_fonction(parametres):
    """
    Ce que fait la fonction
    Parameters
    ----------
    par1 : type de par1
    par2 : type de par2

        Avec une description plus longue.
        Sur plusieurs lignes.

    Returns
    -------
    type1
        Le return associé à type 1

    """
    corps de la fonction
```

Mais la plupart du temps, pour les fonctions de taille moyenne et petites, nous utiliserons plutôt la version courte ci-dessous.

Version courte (à utiliser)

```
def exemple_fonction(parametres):
    """
    parametres -> ce qu'elle renvoie
    """
    corps de la fonction
```

4. OUTIL DE VÉRIFICATION DE CODE

Il existe différentes méthodes pour cela. Je vous conseille de la faire de manière régulière pendant toute la rédaction de votre projet, en utilisant l'outil en ligne suivant :

http://pep8online.com